IMPROVEMENTS AND AUGMENTATIONS TO LEARNING BASED JAVA: A JAVA BASED LEARNING BASED PROGRAMMING LANGUAGE

BY

YIMING JIANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisor:

Professor Dan Roth

Abstract

Machine Learning (ML) is the science that enables computers with the ability to learn without being explicitly programmed. ML is so pervasive today, with applications in speech recognition, recommendation systems, fraud detection and many more that we may not be aware of. To facilitate a rapid pace of development, it is important to create a framework with modularity and reusability. Learning Based Java (LBJava) was introduced by Cognitive Computation Group (CCG) to achieve such goal.

This thesis extends and introduces multiple components in LBJava. We begin by giving a comprehensive literature review relates to Learning Based Programming (LBP) and LBJava.

Then we introduce regression evaluation metrics to LBJava. In addition, we introduce Adaptive Sub-Gradient (AdaGrad) for regression. Then we add a comprehensive tutorial with example on regression. Furthermore, we extend both SGD and AdaGrad algorithms for classification. Then we evaluate across various learning algorithms, with sparse and dense features, using large programmatically generated datasets.

Moreover, we introduce Neural Network (NN), in particular, Multilayer Perceptron (MLP), to LBJava. We also did some miscellaneous work.

Lastly, we conclude on all the extended and added components and provide recommendations for future work.

To my parents, for their love and support.

Acknowledgments

This thesis would not have been possible without the support of many people.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Dan Roth, for his guidance, immense knowledge, caring and assistance.

I would like to give special thanks to Dr. Christos Christodoulopoulos, and without whose help, patience, encouragement and knowledge, this thesis would not have been possible. Also, thanks to Daniel Khashabi, for his help, encouragement and suggestions.

Thanks to the Department of Computer Science for offering me a Teaching Assistantship, providing me with the financial means to complete my Master's degree.

Thanks to the kind souls who have written and edited this LaTeX template and put it online.

Thanks to my friends for their friendship, love and support.

Also, I am thankful to all others who have helped me in the past five years. I would not have been able to complete my Bachelor's degree and Master's degree without the wisdom, experience, encouragement and help from Mr. Morgan Zhang.

Finally, I am grateful to my parents, for their unconditional love, encouragement and support in my most difficult times. I am also grateful for their hard work and support that I was able to pursue college education in the Unites States.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AdaGrad	Adaptive Subgradient
ANN	Artificial Neural Network
CCG	Cognitive Computation Group
EV	Explained Variance
FOL	First Order Logic
LBJava	Learning Based Java
LBP	Learning Based Programming
LMS	Least Mean Squares
MAE	Mean Absolute Error
MedAE	Median Absolute Error
MSE	Mean Squared Error
ML	Machine Learning
MLP	Multilayer Perceptron
NLP	Natural Language Processing
NN	Neural Network
OOP	Object Oriented Programming
RMSE	Root Mean Squared Error
SGD	Stochastic Gradient Descent

Chapter 1

Introduction

1.1 Motivation

Machine Learning (ML) is the science that allows computers to learn without being explicitly programmed. Researchers across various domains are applying ML to solve problems that conventional programming techniques fell short. In conventional programming, we code up a set of rules, feed them into the computer, along with data, and hope it would produce desired results. In contrast, as for ML, we prepare data and information about the desired results. Then we feed them into the computer with some learning algorithm, which in turn would learn a set of rules that would solve our problem.

Learning based programs and systems are ubiquitous today. To ensure a rapid and efficient development of such programs, significant engineering efforts are needed to create a framework with modularity and reusability.

1.2 Learning Based Programming

Learning based program [Roth, 2005] is referred as any program with learning components. And Learning Based Programming (LBP) is referred as the study of learning based program. LBP is a programming paradigm that allows programmers to write program using variables that are not explicitly defined during programming. In chapter 2, we review several papers which introduced and elaborated on this programming paradigm.

1.3 Learning Based Java

Learning Based Java (LBJava) [Rizzolo and Roth, 2007] was the first generation of LBP language, developed by the CCG group. LBJava is a discriminative modeling language that allows programmers to express constraints declaratively in arbitrary First Order Logic (FOL) formulas. LBJava's run-time library translates and generates explicit programs, with Object Oriented Programming (OOP) principles.

Functional programming has gained a significant momentum in recent years, due to its simplicity and expressiveness. Saul [Kordjamshidi et al., 2015], a new generation of declarative programming language, was presented by CCG group. Unlike LBJava, which was written in Java, Saul was implemented by Scala, a functional and object oriented programming language.

1.4 Summary of Contribution

This thesis enriches LBJava in many components and subsequent chapters present the work in similar themes.

Among all existing learning algorithms in LBJava, Stochastic Gradient Descent (SGD) was the only algorithm for regression and all others were for classification. There was a detailed evaluation metric for classification, but there was a lack of evaluation metric for regression. We introduce *TestReal* class, outputting in similar format as *TestDiscrete* to LBJava.

In addition, we introduce AdaGrad learning algorithm for regression, as a comparison to SGD. We extend both SGD and AdaGrad for classification as well.

Moreover, we evaluate across various learning algorithms, with both sparse and dense features, on large programmatically generated datasets.

Furthermore, we introduce Artificial Neural Network (ANN), in particular, Multilayer Perceptron (MLP) to LBJava.

Also, we evaluate and compare across multiple learning algorithms on NLP tasks, i.e. POS Tagging and Chunking.

For miscellany, we fix some bugs and compilation failures and we add some various tutorials and documentations, both from user and developer perspective, to facilitate understanding.

Lastly, we conclude on the contribution and propose several directions for further development.

Chapter 2

Literature Review

2.1 Learning Based Programming

Learning Based Programming (LBP) refers to a programming paradigm that allows programmers to write program using variables that are not explicitly defined during programming. In addition, LBP extends conventional programming to support writing programs that some definitions are generated in a data-driven way, or learned from examples and observations upon executing [Roth, 1999]. The definitions of variables in LBP, and the internal interactions, are generated in a data-driven way, straight from the source of the data. It provides the developers, to set up definitions only by declaring the target concepts, and the source of information, that would contribute to the definitions. On the other hand, the concrete concepts and variables, are learned and generated as more examples fed into the system.

Several notions are essential in LBP, serving as fundamental building blocks for the theory. *Relational variable* is a relation mapping, from instance to its truth value. *Structural instance space* is a graph, constructed by the system, from sensed elements. Thus, *Relation generation function* generates a concrete variable, when the instance is present. It defines uniformly across different variables that are coming from data-driven method, or learned from examples and observations.

Moreover, an LBP program consists of multiple programs, where each program is determined by a process of data driven compilation. It is a chain of programs, creating a pipeline, from the data source to the target concepts.

The steps in LBP paradigm lay a solid fundamental theoretical foundation, for many ML system today. It makes developers to develop large scale systems, with the bulk of knowledge, coming from learning from raw data, and acts robustly, on unseen data.

2.2 Learning Based Java

Learning Base Java (LBJava) [Rizzolo and Roth, 2010], [Rizzolo and Roth, 2007] is the first generation of the implementation, based off the LBP programming paradigm. LBJava is a discriminative modeling language that allows programmers to express constraints declaratively in arbitrary First Order Logic (FOL) formulas. LBJava's run-time library translates and generates explicit programs, with Object Oriented Programming (OOP) principles.

In LBJava, a model represents an objective function that weight vector is implicit. Features, labels and constraints are specified in a special syntax, as discussed in detailed in Section 3.2. Thus, each instance of an model contains its own weight vector. The pipeline process of LBJava is discussed in detail in Section 3.1. One notable feature is that feature extraction, learning and inferencing are all truly on-line, which means the function's internals and result are performed on demand when it is invoked by the internal Java code. This decision of on-line fashion makes developers, especially, when multiple learning functions are involved, more convenient. The downside is that the amount of memory used is increased, because we need to hold more to process in the pipeline.

The introduction of LBJava, enables ML to be accessible by hiding feature extraction, learning, and inferencing from developers as much as possible. This ease of work load would allow developers to focus on their own domain problem, rather than working on building and verifying the ML system.

2.3 Saul

Saul [Kordjamshidi et al., 2015] is the next generation of LBP, based on LBJava. Saul is implemented in Scala, an object-functional programming language, which has the advantage of simplicity, expressiveness in functional programming languages, yet remains the Object-Orient Programming (OOP) principles. Saul facilitates developers to learn, name and manipulate named abstractions on relational data and make decisions respect domain constraints, with a level of inference. Saul allows developers to create a ML system to solve their problems, involving minimal coding, that conventional programming languages would require much more.

Chapter 3

LBJava Basics

3.1 Pipeline

There are multiple layers of abstraction in LBJava, with stages of compilation and code generation. The intent is to provide an elegant interface for users to use ML algorithms to solve problems.

The layer that users need to have a direct interaction is the `lbj` layer. `lbj` file has a special, yet simple syntax, to define features, labels, classifiers and how to train and test the data set. The LBJ syntax is thoroughly discussed in Section 3.2. The syntax involves only a few keywords with several arguments and the remaining code is exactly Java code. In addition, in fact, users can define more than one feature and use them in the classifier. In the definition of the classifier, users can specify the training data set, the learning algorithm with specific parameters, testing data set and the granularity of the output printing out onto the console.

However, the users still need to explicitly specify in Java code for feature extraction. A class inherited from `parser` needs to be implemented by users, on how data set is parsed, to features and label, respectively. Typically, the features are parsed from the parser, in the form of a list, or an array. We iterate through the list or the array and use the syntax keyword `sense` for each entry.

Once `lbj` file is completed by users, LBJava has a parser that reads the definitions of the features, labels and classifiers and generated the corresponding Java classes, yet sharing the same parent class `Classifier`.

Next step in the pipeline is the training phrase. Using the definition of classifier declared in `lbj`, a `BatchTrainer` instance is internally invoked and it would train the classifier for the number of iterations, as specified in the definition of the classifier. Each example is fetched from the parser, via interface `public Object next()` from `parser` and is fed into the learning classifier sequentially.

The last stage in the pipeline is the evaluation phrase. Similar to the training phrase, where a parser parses the testing data into the classifier. Internally, a testing metrics class takes the output from the classifier and compares against the label, outputting both system information and evaluation information.

3.2 LBJ Syntax

The syntax of LBJ syntax is initially developed, as part of LBJava implementation, in [Rizzolo, 2011].

3.2.1 Classifier Declarations

This section defines the syntax, of classifier precisely, which involves a combination of keyword expressions and Java code. In LBJava, both the definitions of feature and label fall into the domain of classifier.

Classifier declarations are used to name classifier expression and the syntax of a classifier declaration has the following form:

```
feature-type classifier-name (type name) <- {
        classifier-expression-body
}
```

`feature-type` is the return type of the classifier. Possible keywords for the feature type are: `real`, `real%`, `real[]` and `discrete`, `discrete%`, `discrete[]`. The keyword `real` stands for the continuous data, as the keyword `discrete` is for the categorical data. The operator % indicates that the same feature value is mapped to the same index, as for the operator [] maps duplicate feature values to a new index. Supposing, the feature that we use is bag of words and the data set we use are articles. Each word is mapped into an index of word ID. If the data set contains the following sentence: "Bluewater computer is a computer". "Bluewater" is the first word that we see, thus, it gets the index mapping of 0. Similarly, "computer" is the word that we have never seen before, thus, it gets the index mapping of 1. If we use operator %, the second "computer" still gets the index mapping of 1, due to the duplication. On the contrary, if operator [] is used, the second "computer" gets the index mapping of 5.

`type name` is the Java class object parsed in. Inside the `classifier-expression-body`, `sense` statement is used, to instantiate a primitive feature that has been detected when computing an array of features.

Bag of words feature as an example of `lbj` definition is shown below.

```
discrete% BagOfWords(Document d) <- {
        List words = d.getWords();
        for (int i = 0; i < words.size(); i++)
                sense words.get(i);
}
```

In the example definition above, `type name` is `Document d`, and we iterate through all the words in the document, `sense` each word in the list.

The label for classification definition is shown below as an example.

```
discrete NewsGroupLabel(Document d) <- {
        return d.getLabel();
}
```

3.2.2 Learning Classifier Expressions

The learning classifier defines the features classifier, label classifier, parser, learning algorithm, training, testing, cross validation, and tuning. It has the following expression syntax:

```
learn labeler-expression
using feature-extractor-expression
from parser-expression
with learning-algorithm-expression {
}
cval [int] split-strategy
testFrom parser-expression
progressOutput [int]
```

In the example above, `learn` keyword takes one argument, the definition of label classifier. `using` keyword takes multiple arguments, the definitions of feature classifiers. `from` keyword takes one argument, the parser, taking the training data set. `with` takes the name of the learning algorithm, and inside the braces, parameters configured for the learning algorithm in Java code are defined. `cval` statement, is the k-fold cross validation, which helps to prevent overfitting problem. The first argument is the k and the second argument is the strategy, such as, "random". `testFrom` takes the parser, from the testing data set. Lastly, `progressOutput` is the granularity of the number of the examples printed onto the console.

An example usage of the learning classifier, for newsgroup scenario is shown below.

```
discrete NewsGroupClassifier(Document d) <-
        learn NewsGroupLabel
        using WordFeatures, BigramFeatures
        from new DocumentReader("data/20news/train")
        5 rounds
```

```
        with SparseNetworkLearner {
                SparseAveragedPerceptron.Parameters p =
                new SparseAveragedPerceptron.Parameters();
                p.learningRate = 0.05;
                p.thickness = 5;
                baseLTU = new SparseAveragedPerceptron(p);
        }
        testFrom new DocumentReader("data/20news/test")

        progressOutput 2000
end
```

3.2.3 Parameter Tuning

Parameter tuning is essential in ML, as it allows the programmers to find the best parameters in the learning algorithms to perform to their maximum extent.

LBJava has two syntax expressions, for parameter tuning. The first type is the set of parameters. Suppose we would like a parameter to try a set of predefined values, the syntax is the following:

```
{{value1, value2, value3}}
```

For example, we want the algorithm to try 5, 10, 20, 30, 40 iterations.

The `lbj` declaration would look like this:

```
{{5, 10, 20, 30, 40}} rounds
```

The second type of syntax is for stepwise parameters, within a range, of a step size.

Let `start`, `end`, and `step_size` to denote the start of the range, end of the range and the step size, respectively. The syntax for stepwise parameter tuning looks like:

```
{{step_size -> start : end}}
```

As an example, wa would like to try thickness in `SparseAveragedPerceptron`, from 3 to 0.5, with step size of 1.

The `lbj` declaration would look like this:

```
p.thinkness = {{ 1 -> 3 : 0.5}};
```

3.3 SNoW Architecture

LBJava borrowed the architecture idea from SNoW [Carlson et al., 1999], which consists of a sparse network of linear threshold units. The architecture maintains a two-layer network. The first layer is the input layer, also is the feature layer. Nodes are instantiated for features observed in the training example. The second layer contains target nodes, and each node corresponds to a class label.

SNoW and LBJava use the idea that for each element, i represents that the ith feature is active and the inactive features will not be in the array [Blum, 1990]. There are connections between nodes in the first layer and in the second layer. The connections carry the weights between nodes. Also, these connections are allocated dynamically: a feature f is allocated and linked to target node t, if and only if t is present in the example, with label t. Thus, the first pass of the data set construct the network, and new negative examples are not fed into classifiers have past.

In LBJava, `SparseNetworkLearner` is implemented in this architectural idea. Precisely, there is an classifier for each label. All active feature with such label are fed into the classifier as positive examples and all other labels are treating as negative examples. Upon prediction time, winner-takes-all policy is deployed, to assign the label. Inside `SparseNetworkLearner`, there is a list of `LinearThresholdUnit` classifier, for each unique label.

Originally in LBJava, `SparseNetworkLearner` is used only for multi-class classification. In fact, even for binary classification tasks, `SparseNetworkLearner` needs to be used, regardless of what internal algorithms to use, i.e. `SparseAveragedPerceptron`, `SparseWinnow`. Examples in LBJava for binary classification, such as Spam, use learning algorithm directly. This is incorrect, given the previous discussion. Thus, updates on the corresponding `lbj` files have been made. This big issue is realized recently, thus, documentation is added to the code repository, to make special notice of instructions that learning algorithms should not be used directly. They all need to be used inside `SparseNetworkLearner`.

An example below demonstrates how to use `SparseWinnow` correctly.

```
with SparseNetworkLearner {
        SparseWinnow.Parameters p = new SparseWinnow.Parameters();
        p.learningRate = 1.1;
        p.beta = 0.909;
        p.thickness = 2;
        baseLTU = new SparseWinnow(p);
}
```

Chapter 4

Regression Evaluation

4.1 Motivation

In ML, model evaluation is crucial, as it quantifies the quality of predictions. *TestDiscrete* class is the model evaluation class in LBJava for classification. *TestDiscrete* outputs both system metrics, such as memory usage and evaluation execution time, and evaluation metrics, such as precision, recall, F1, label count, prediction count for each label and overal accuracy.

Our initial intention was to introduce AdaGrad algorithm into LBJava. Since AdaGrad is a variation of SGD, it would be interesting to compare between AdaGrad and SGD. SGD was implemented originally for regression and there was a lack of model evaluation metrics for regression. It is straightforward to introduce a class to evaluate regression prediction quality, which is named *TestReal*.

The following function signature is a static method in *TestReal* class to do the evaluation and output both system and statical metrics.

```
public static void testReal(TestReal tester,
                            Classifier classifier,
                            Classifier oracle,
                            Parser parser,
                            boolean output,
                            int outputGranularity)
```

An instance of *TestReal* is necessary to pass in to isolate internal data book keeping when *TestReal* is used for another evaluation. The classifier named *classifier* is the model that we evaluate on. The classifier named *oracle* is the gold label, reading from the label of testing set. An parser is passed to iterate through the testing set. Boolean flag *output* is used to decide whether the output would printed on *standard out*. *outputGranularity* is a parameter to specify the amount of example processed to output onto *standard out*. This value should set with minimum value of 1.

4.2 System Metrics

TestReal outputs system metrics consistently with *TestDiscrete*, where the start execution time and memory usage, the processing time and memory usage for the first example, time stamps for number of example being processed, according to *outputGranularity* and average evaluation time.

A sample output for system metrics is shown below.

```
0 examples tested at Fri Mar 25 23:17:51 CDT 2016
Total memory before first example: 356515840
First example processed in 0.001 seconds.
Total memory after first example: 356515840
10 examples tested at Fri Mar 25 23:17:51 CDT 2016
20 examples tested at Fri Mar 25 23:17:51 CDT 2016
30 examples tested at Fri Mar 25 23:17:51 CDT 2016
40 examples tested at Fri Mar 25 23:17:51 CDT 2016
50 examples tested at Fri Mar 25 23:17:51 CDT 2016
60 examples tested at Fri Mar 25 23:17:51 CDT 2016
70 examples tested at Fri Mar 25 23:17:51 CDT 2016
80 examples tested at Fri Mar 25 23:17:51 CDT 2016
90 examples tested at Fri Mar 25 23:17:51 CDT 2016
100 examples tested at Fri Mar 25 23:17:51 CDT 2016
106 examples tested at Fri Mar 25 23:17:51 CDT 2016

Average evaluation time: 9.433962264150944E-6 seconds
```

4.3 Statistical Metrics

There are several statistical metrics implemented in *TestReal*, including: Root Mean Squared Error (RMSE), Mean Squared Error (MSE), Mean Absolute Error (MAE), Median Absolute Error (MedAE), Explained Variance (EV) and R2 Score. The statistical computation uses Apache Common Math [Foundation, 2004] library.

y_i denotes the ith example of the gold value and $\hat{y}_i$ denotes the predicted value for the ith example. n denotes the total number of testing examples.

RMSE is one of the most frequently used measure of the differences between values predicted by a model

and the actual values. RMSE represents the sample standard deviation of the differences between predicted values and actual values. RMSE is a single prediction measure, with aggregations from prediction errors in all times.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y_i} - y_i)^2}{n}}$$

Similarly, MSE is a measure on the quality of a model which is the square of RMSE.

$$MSE = \frac{\sum_{i=1}^{n}(\hat{y_i} - y_i)^2}{n}$$

MAE is a risk metric corresponding to the expected value of the absolute error loss.

$$MAE = \frac{\sum_{i=1}^{n}|\hat{y_i} - y_i|}{n}$$

MedAE is particularly interesting because it is robust to outliers due to the median nature.

$$MedAE = median(|y_1 - \hat{y_1}|, \ldots, |y_n - \hat{y_n}|)$$

EV is a measure on the proportion of a model, accounts for the variation of a given data set. The best possible value is 1.0 and lower values are worse.

$$EV = 1 - \frac{Variance((y_1 - \hat{y_1}), \ldots, (y_n - \hat{y_n}))}{Variance(y_1, \ldots, y_n)}$$

Lastly, R2 Score, also known as the coefficient of determination, measures how well future samples would be predicted by the model. Best possible value is 1.0. An R2 score of 1 indicates that the regression line fits the data perfectly and an R2 score of 0 indicates that the regression line does not fit the data at all because data is more non-linear than the curve, or the data is random. R2 score can be negative, because a model can be arbitrarily worse.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where

$$\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

Chapter 5

Introduce AdaGrad

5.1 Stochastic Gradient Descent

We first introduce the notion of loss function and a general optimization method, gradient descent. Then we propose an improved approach, stochastic gradient descent, based on gradient descent.

5.1.1 Loss Function

Loss function qualifies the amount by which the predicted values deviates from the actual ones. In ML, classification and regression are formulated as the minimization problem of a loss function over a training data set.

Least Mean Squares (LMS) is one of the common loss functions for regression.

$$J_{LMS}(y, x, w) = \min_{w} \frac{1}{2} \sum_{i=1}^{m} (y_i - w^T x_i)^2$$

Hinge loss is a loss function most notably for Support Vector Machine.

$$J_{Hinge}(y, x, w) = max(0, 1 - yw^T x)$$

5.1.2 Gradient Descent

Gradient Descent [Shalev-Shwartz and Ben-David, 2014] is a general strategy for minimizing the objective function $J(w)$. Since the gradient is in the direction of the steepest increase in the function, to get to the minimum, we go in the opposite direction. The general gradient descent algorithm is shown in Algorithm 1.

The gradient of $J(w)$ for LMS is computed as:

$$\frac{\partial J}{\partial w_j} = -\sum_{i=1}^{m} (y_i - w^T x_i) x_{i,j}$$

Algorithm 1 Gradient Descent Algorithm

Input: *loss function* $J(w)$

Start with an initial guess for w, denoted as $w^{(1)}$

for $t = 0, 1, 2, \ldots$ **do**
  Compute the gradient of $J(w)$ at w^t, denoted as $\nabla J(w^t)$

  // Update w^t to get w^{t+1} by taking a step in the opposite direction of the gradient
  // η: learning rate
  $w^{(t+1)} = w^{(t)} - \eta \nabla J(w^t)$
end for

5.1.3 Stochastic Gradient Descent

In the gradient for LMS, we see that the weight vector is not updated until all errors are calculated, which is called the batch mode of gradient descent. Instead, another approach, named Incremental or Stochastic Gradient Descent (SGD) makes updates to the weight vector as soon as encountering errors, rather than waiting for a full pass over the training data set. Batch Gradient Descent is described in Algorithm 2, as compared to SGD, in Algorithm. SGD can approximate Batch Gradient Descent arbitrarily close, if η is small enough.

Algorithm 2 Batch Gradient Descent

Input: *loss function* $J(w)$, *training data set* D

while Not Satisfied **do**
  Compute the gradient $\nabla J_D(w)$
  $w \leftarrow w - \eta \nabla J_D(w)$
end while

Algorithm 3 Stochastic Gradient Descent

Input: *loss function* $J(w)$, *training data set* D

while Not Satisfied **do**
  for *training example* d in D **do**

    Compute the gradient $\nabla J_d(w)$
    $w \leftarrow w - \eta \nabla J_d(w)$
  end for
end while

The Online/Incremental Gradient Descent, SGD, is often preferred, when the training set is very large. It may get close to the optimum much faster than the batch version of Gradient Descent. The other advantage of SGD is that it is an efficient algorithm that can be implemented easily with a few lines of code.

`StochasticGradientDescent` learner was implemented in LBJava, but it was for regression scenario.

Also, it uses LMS as loss function.

5.2 Adaptive Subgradient

Although SGD is an advancement from Batch Gradient, we use fixed learning rate η in SGD, but this can change. Adaptive SubGradient (AdaGrad) was proposed [Duchi et al., 2011] which alters the update rule to adapt based on historical information, so that frequently occurring features in the gradients get smaller learning rate and infrequent features get larger ones. Essentially, the idea is to learn slowly from frequent features, but pay attention to rare but informative features.

The key idea is to define a per feature learning rate for the jth feature, in the tth iteration, as:

$$\eta_{t,j} = \frac{\eta}{\sqrt{G_{t,j}}}$$

where

$$G_{t,j} = \sum_{k=1}^{t} g_{k,j}^2$$

which is the sum of squares of gradients of feature j until time t.

The update rule for AdaGrad is:

$$w_{t+1,j} = w_{t,j} - g_{t,j}\eta_{t,j}$$

The advantage of AdaGrad is that it is easy to implement and it tends to work well in practice. AdaGrad usually updates weights faster than Perceptron or SGD. Lastly, AdaGrad is not sensitive to the initial learning rate η_o.

5.3 AdaGrad Regression Learner

`AdaGrad` learner class, for regression, is added into LBJava. It extends the abstract `Learner` class and uses the `public void learn()` interface, from `Learner`, to take feature indices, feature values, label indices and label values as arguments.

We provide `AdaGrad` with two loss functions, namely, LMS and hinge loss. The default option is hinge loss, but it can be configured via the parameter class, as shown in the code snippet below.

```
AdaGrad learner = new AdaGradClassifier();
```

```
AdaGrad.Parameters p = new AdaGrad.Parameters();
p.learningRateP = 1;
p.lossFunctionP = "lms";
learner.setParameters(p);
```

From Algorithm 3, we can see that, the update rule for weight vector is:

$$w_{t+1} = w_t - \eta g_t$$

where g_t is the gradient vector.

For LMS loss function, the gradient is:

$$g_t = (w_t * x_t - y_t)x_t$$

For hinge loss function, the gradient is:

$$g_t = -y_t x_t$$

In addition, the update rule is slightly different. For hinge loss, we only update, when a mistake is made. The criteria, for a mistake, is if $y_i w_i * x_i \leq 1$ where 1 is the margin for the mistake.

As for regression, function `public String getOutputType()` from `Learner` needs to be overridden to return `"real"`, rather than the default value `"discrete"` in `Learner`. This data type needs to be known upon lbj parser reads the lbj file.

In testing phrase, testing feature data, i.e. feature indices and feature values, parses into an interface `public FeatureVector classify()`, from `Learner`. The regression computation lies in interface `public double realValue()`, to calculate $wT + \theta$.

5.4 Compare AdaGrad and SGD on Regression

We compare AdaGrad and SGD on regression on Bike Sharing Data Set [Fanaee-T and Gama, 2014] maintained from University of California, Irvine [Lichman, 2013]. There are two sets of data sets from this Bike Sharing Data Set, namely Hour and Day. Detailed information on the number of features, number of instances and what each feature means, for Hour and Day data sets, respectively, can be found in Appendix A.

The evaluation results for Day data sets are shown in Table 5.1 and Table 5.2, and for Hour data sets are shown in Table 5.3 and Table 5.4.

We analyze the comparison between AdaGrad and SGD from three perspectives: performance, convergence and pick of learning rate.

Performance

As shown in Table 5.1, and Table 5.3, running with 10^4 iterations on Day data sets and running with 10^3 iterations on Hour day sets, shows that AdaGrad and SGD have comparable performance. Although AdaGrad is slightly better, with even more iterations, AdaGrad and SGD will reach complete convergence to give equivalent output performance, in terms of statistical metrics.

In terms of statistical metrics, as discussed in Section 4.3, RMSE, MSE, MAE and MAE are all close to 0, indicating that the difference between the prediction and the target value is fairly close. In addition, EV and R2 score are all 1s, indicating that the model would predict very well, on future samples.

Convergence

AdaGrad uses a dynamic learning rate, such that the learning algorithm learns slowly from examples appear more often, and learn at a faster rate, from examples appear less often. On the contrary, SGD uses a constant learning rate, for all examples. We demonstrate the effect of the dynamic learning rate, by comparing AdaGrad and SGD on the same data set, using different number of iterations, with the same initial learning rate. As shown in Table 5.1 and Table 5.2, for AdaGrad, the performance is comparable between running with 10^3 and 10^4 iterations, meaning that AdaGrad reaches the convergence stage, at about 10^3th iterations. However, for SGD, there is a dramatic gap of difference, in terms of performance, between running with 10^3 and 10^4 iterations, meaning that SGD reaches convergence much later than AdaGrad. Similarly, we can show that AdaGrad reaches convergence faster than SGD, from Table 5.3 and Table 5.4.

Pick of Learning Rate

As shown in Table 5.1, Table 5.2, Table 5.3, and Table 5.4, AdaGrad uses the same initial learning rate, 1, even for different data sets. However, SGD uses a different initial learning rate, for each data set. Again, with the dynamics in the learning rate in AdaGrad, it can gradually find the converging optimal learning rate, given more examples. Thus, the pick of initial learning rate for AdaGrad is not quite import, yet, it is very critical to pick the appropriate initial learning rate for SGD. Therefore, it is easier to tune AdaGrad than SGD.

Table 5.1: AdaGrad vs SGD on Bike Sharing Day Data Set for Regression -1

Algorithms		
Statistical Metrics	AdaGrad	SGD
	LearningRate = 1 Iterations = 10^4	LearningRate = 10^{-11} Iterations = 10^4
Root Mean Squared Error	0.000109	0.009048
Mean Squared Error	0	0.000082
Mean Absolute Error	0.000088	0.006933
Median Absolute Error	0.00008	0.005834
Explained Variance	1	1
R2 Score	1	1

Table 5.2: AdaGrad vs SGD on Bike Sharing Day Data Set for Regression - 2

Algorithms		
Statistical Metrics	AdaGrad	SGD
	LearningRate = 1 Iterations = 10^3	LearningRate = 10^{-11} Iterations = 10^3
Root Mean Squared Error	0.020947	54.786436
Mean Squared Error	0.000439	3001.553619
Mean Absolute Error	0.017300	39.905649
Median Absolute Error	0.014259	27.480436
Explained Variance	1	0.999270
R2 Score	1	0.999266

Table 5.3: AdaGrad vs SGD on Bike Sharing Hour Data Set for Regression - 1

Algorithms		
Statistical Metrics	AdaGrad	SGD
	LearningRate = 1 Iterations = 10^3	LearningRate = 10^{-8} Iterations = 10^3
Root Mean Squared Error	0	0.002464
Mean Squared Error	0	0.000006
Mean Absolute Error	0	0.001917
Median Absolute Error	0	0.001524
Explained Variance	1	1
R2 Score	1	1

Table 5.4: AdaGrad vs SGD on Bike Sharing Hour Data Set for Regression - 2

Algorithms		
Statistical Metrics	AdaGrad	SGD
	LearningRate = 1 Iterations = 10^2	LearningRate = 10^{-8} Iterations = 10^2
Root Mean Squared Error	0.000110	0.258432
Mean Squared Error	0	0.066787
Mean Absolute Error	0.000075	0.210825
Median Absolute Error	0.000059	0.177578
Explained Variance	1	0.999999
R2 Score	1	0.999998

5.5 SGD and AdaGrad Classification Learners

The learning algorithms in SGD and AdaGrad, with either hinge loss or LMS loss functions, for classification and regression are exactly the same. However, in terms of the actual implementation, there are some minor differences.

Firstly, the inheritance structure is different. SGD and AdaGrad regressors are inherited directly from `Learner`. On the other hand, SGD and AdaGrad classifiers are wrapped inside `SparseNetworkLearner`, thus they inherit from `LinearThresholdUnit` instead.

Secondly, in regression, `learn` interface takes the value of the label as the y to compute. On the contrary, in classification, the index of the label is used, rather than the value. Due to the SNoW architecture, label index of 1 is seen as positive example and label index of 0 is seen as negative example.

Thirdly, the output is generated differently. In regression, the output is computed as $wT * x + \theta$, while, in classification, label indices are computed via $wTx + \theta$ comparing with threshold. An instance array, `predictions`, from `Learner` stores all the prediction mapping from labels to indices.

5.6 Compare AdaGrad and SGD on Classification

We evaluate AdaGrad and SGD on classification on Monotone Data Set in Appendix B. The data set is the same in Table 5.5 and Table 5.6, with parameters: $l = 10, m = 100, n = 500, k = 50000$.

We compare AdaGrad and SGD on classification, with both hinge loss and LMS loss functions, under conditions of two different iterations. Similarly, we compare across the following perspectives: performance, hinge vs lms, convergence, and pick of learning rate.

Performance

As shown in Table 5.5 and Table 5.6, with 10 iterations, both AdaGrad and SGD, with two different loss functions, achieve 100% accuracy, indicating that the performance is comparable, with tuning of the appropriate initial learning rate.

Hinge vs LMS and Convergence

As shown in Table 5.5, with only 2 iterations, for AdaGrad, since it can dynamically adjust learning rate for features, the performance is comparable, across hinge loss and LMS loss. However, for SGD, with the same initial learning rate of 10^{-3}, LMS loss outperforms hinge to some extent. However, with initial learning rate

Table 5.5: AdaGrad vs SGD on Monotone Data Set - 1

Algorithms		
Loss Functions	AdaGrad	SGD
	LearningRate = 0.1 Iterations = 2	LearningRate = 10^{-3} Iterations = 2
Hinge Loss	99.96	92.39
LMS Loss	99.66	97.86

Table 5.6: AdaGrad vs SGD on Monotone Data Set - 2

Algorithms		
Loss Functions	AdaGrad	SGD
	LearningRate = 0.1 Iterations = 10	LearningRate = 10^{-3} Iterations = 10
Hinge Loss	100	100
LMS Loss	100	100

of 0.01, LMS has only 50.42% of accuracy, yet hinge loss has 99.15%, which demonstrate that hinge loss has the capability to learn more quickly than LMS, with a potential to reach convergence faster.

Pick of Learning Rate

Since AdaGrad has the ability to dynamically change the weight of the learning rate, similar conclusion can be made that the initial learning rate does not matter very much in AdaGrad, which, reduces the complexity and eases the burden of tuning on learning rate. On the contrary, for SGD, a careful chosen learning rate is necessary. A large learning rate would make SGD miss the local minimum and a small learning rate would lengthen the number of iterations before reaching convergence.

Chapter 6

Introduce Neural Network

6.1 Neural Network

Neural Network (NN) was originally inspired by human brains and they are algorithms try to mimic how human brain works. NN was very widely used in the 1980s and early 1990s and the popularity diminished in the late 1990s. However, there is a recent resurgence of NN. Since NN is computationally expensive, with the advancement in computation power in the past decades, large scale neural networks became computationally feasible.

Human brain consists of a network of neurons. At a simplistic level, neuron is essentially a computational unit, that gets a number of inputs from its input wires, called dendrite, does some computation and sends the outputs to its output wires, called axon. Artificial Neural Network (ANN) is usually referred as the model in NN. Similarly, a neurone in ANN is a logistic unit, which has input wires taking inputs, has a logistic unit that does computation, and sends the output down via output wires. A set of neurons are packed into one layer and the network consists of interconnections between neurons, from different layers.

A typical ANN consists of three parameters. The first one is the interconnection patterns between different layers of neurons. The second one is the learning process which governs the rules to update weights of the interconnections. The last one is the activation function that computes the weighted input of a neuron to its output activation.

6.2 Multilayer Perceptron

Multilayer Perceptron (MLP) is a feedforward ANN model, where connections between neurons does not form a cycle, as opposed to recurrent neural network. MLP consists of multiple layers of neurons, with each layer fully connected to the next one. Typically, the first layer is called the input layer, which has input nodes and the last layer is named the output layer, which has a set of output nodes. The layers between input layer and the output layers are called hidden layers. There could be no, one or more hidden layers.

Each node in the network is a neuron with nonlinear activation function, except for the input nodes. What distinguish MLP from the standard perceptron learning algorithm is the nonlinear activation function. Suppose all neurons have a linear activation function, it can be shown, by linear algebra, that any number of layers can be reduced to the standard two-layer input-output perceptron model.

Two most common activation functions are both sigmoid functions. The first function is the hyperbolic tangent function, shown below. The second function is the logistic function, also shown below.

$$g(z) = \tanh(z) \qquad \text{Hyperbolic Tangent Activation Function}$$

$$g(z) = \frac{1}{1 + e^{-z}} \qquad \text{Logistic Activation Function}$$

MLP learns through an algorithm called backpropagation, which calculates the gradient of a loss function with respect to all the weights in the network. Optimization methods, such as gradient descent, takes gradient as input and use it to update the weight, attempting to minimize the loss function.

A high level description on the algorithm is in the following. There are two phases in the backpropagation learning algorithm. The first phase is propagations: forward and backward propagations. Training inputs are forward propagated through the network, to generate the output's activations. Then the output's activations are backward propagated through the network, using the training output target, to generate the differences between the actual output values and targeted ones, denoted as deltas, in all output neurons and hidden neurons. The second phase is weight update. For the weight on each connection, between neurons, multiply its output delta and input activation to get the gradient of the weight. Then a ratio, called learning rate, of the gradient is subtracted from the weight. The backpropagation algorithm [Mitchell, 1997] is shown below as Algorithm 4.

6.3 Multilayer Perceptron Learner in LBJava

To introduce MLP learner into LBJava, we want to provide an interface that is consistent with all existing learning algorithms and use it in the same way as any other existing learning algorithms in LBJava, such as `SparseAveragedPerceptron` or `SparseWinnow`.

Algorithm 4 Backpropagation Algorithm

Input: *Network* N, *training data set* D

Initialize all weights to small random numbers.

while Not Satisfied **do**
  for *training example* d in *training data set* D **do**

    Input d into N and compute the network outputs $\vec{o}$

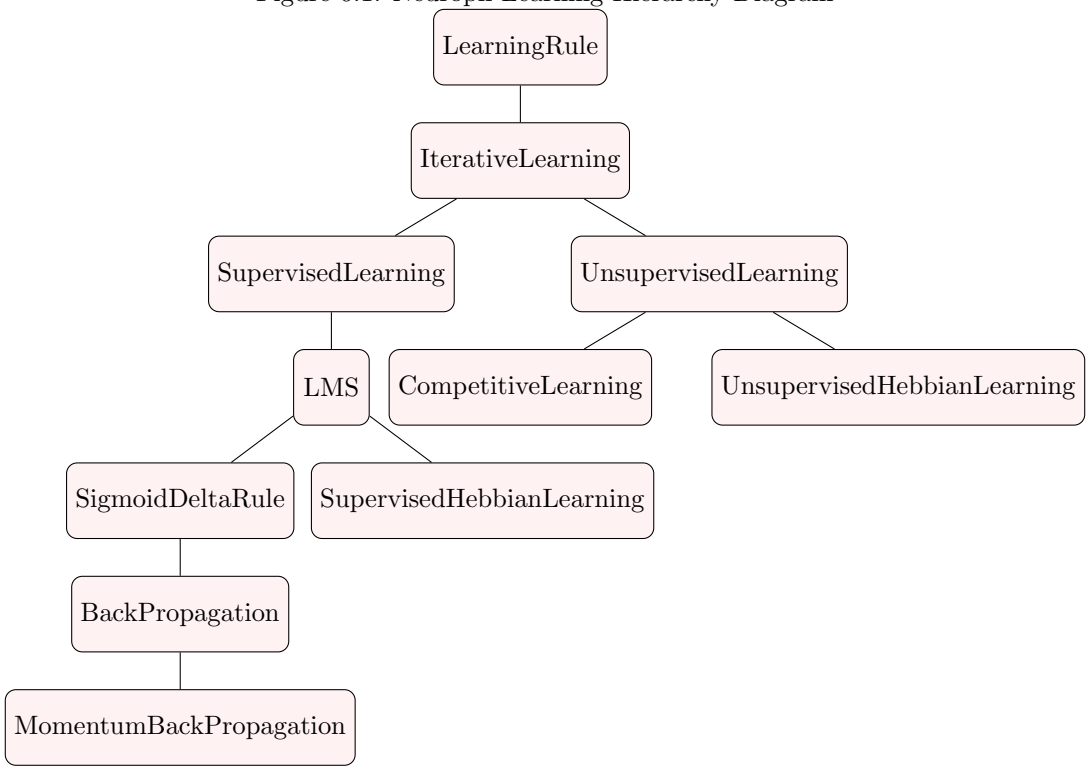
    for each *output neuron* k **do**
      // δ: error; t: target value; o: network output
      $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
    end for

    for each *hidden neuron* h **do**
      // w: weight between hidden neuron and output neuron
      $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$
    end for

    // Update weight
    for each *network weight* $w_{i,j}$ **do**
      // η: learning rate; $x_{i,j}$: the ith feature of the jth example
      $w_{i,j} \leftarrow w_{i,j} + \eta\delta_j x_{i,j}$
    end for

  end for
end while

Figure 6.1: Neuroph Learning Hierarchy Diagram



6.3.1 Introduction to Neuroph Framework

For the actual implementation of MLP, we use an open source NN framework, `Neuroph` [Foundation, 2004], in Java language.

There are several fundamental classes in `Neuroph`, such as NeuralNetwork, Layer, Neuron, Connection, Weight, and LearningRule. These basic components are fully connected, to construct the MLP neural network. Moreover, `Neuroph` has a deep tree of learning hierarchy, for many different types of learning algorithms. We have shown the class diagram in Figure 6.1 to ease the complexity.

6.3.2 Multi-class Classfication in Multilayer Perceptron

Multi-class classification refers to the task of distinguishing between more than two categories. To enable MLP to do multi-class classification, we adopt the one-vs-all scheme. For example, if our MLP is given images as input, the classification task is to tell what is in the image. Suppose there are four possibilities: pedestrian, car, motorcycle, and truck, hence, there are four nodes in the output layer, representing each category in the order of pedestrian, car, motorcycle and truck. If one image has car in it, the output nodes have values $(0, 1, 0, 0)$ where all nodes have 0, indicating the output bit is off, except the car node, having 1,

indicating on state.

6.3.3 MultiLayerPerceptron Learner

The learning algorithm of MLP introduced is named `MultiLayerPerceptron`. It inherits from the `Learner` class, which is an abstract class and the root class for all learning algorithm classes. Therefore, *MultiLayerPerceptron* behaves the same as other learning algorithms.

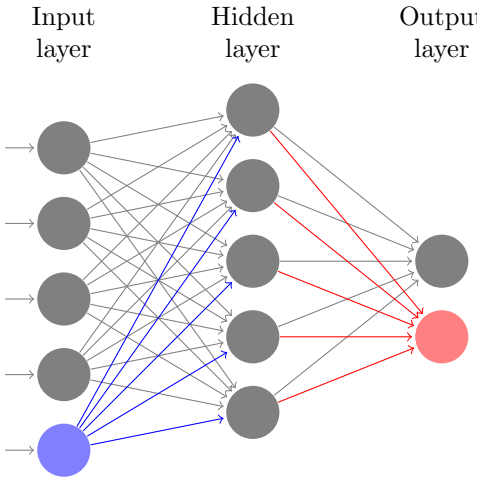
In terms of configuring parameters for MLP, we provide an interface to set three parameters: learning rate, number of hidden layers and number of neurons in each hidden layer. Number of hidden layers and number of neurons in each hidden layers are represented in an array of integers. The length of the array denotes the number of hidden layers and each number in the array denotes the number of neurons corresponding to the layer in the index of the number. The code snippet below demonstrates the usage on setting these parameters.

```
BrownClassifier brownClassifier = new BrownClassifier();
MultiLayerPerceptron.Parameters p = new MultiLayerPerceptron.Parameters();
p.learningRateP = 0.2;
p.hiddenLayersP = new int[] {18, 18};
brownClassifier.setParameters(p);
```

`brownClassifier` is a class extends from `MultiLayerPerceptron`, using the features and labels that we have defined. In the sample code snippet above, the learning rate is set to 0.2 and there are two hidden layers, with 18 neurons in each layer.

In terms of the training process, an interface `learn` function is provided from `Learner` class. The parameters feed into this function are: an integer array of feature indices, a double array of feature values, an integer array of label indices and a double array of label values. In LBJava's pipeline, the first step is to create a custom parser class, extending `Parser` class, on how to parse the data set and how to partition into features and label vectors. The second step is to declare features, label and classifier with the desired learning algorithms in LBJava's syntax in a `.lbj` file. Then a compilation parser in LBJava generates all features classes, label class, and classifier class. Lastly, `BatchTrainer` class takes the parser and the classifier to start the training process. Upon seen an example parsed from the parser, the abstract class `Learner`, hashes each lexicon, or feature, generating the indices of feature vector. The value of the feature vector is 1 indicating that the particular feature is active in the given example. Similarly, each label is also being hashed, outputting the indices of label vector. 1 in label value shows the label is active. These four arrays are generated from `Learner` and fed into the `learn` function in `MultiLayerPerceptron`.

Figure 6.2: Dynamic Adding Input and Output Neurons



In LBJava' scheme, the learning is in online pattern, which means examples are processed one by one in the learning phrase. However, in `Neuroph` implementation, data sets are read into a batch instance first and the learning takes place in online fashion. There is no existing interface in `Neuroph` to provide sequential learning. Thus, we add an abstract interface `abstract public void learn(DataSetRow dataSetRow)` in the abstract class `LearningRule` to take an example instance as argument.

Due to the online scheme of the learning process, we may not know the total number of features or the total number of label categories. We need to dynamically expand the NN, if we see more features, or labels in examples later. For the first example that we see, we construct the NN, where the input neurons correspond to all features and one output neuron to represent the only label. If there are more features, we need to add more input neurons. Similarly, if there are more labels, we need to add more output neurons. Then we need to connect these new neurons to all neurons in the next or previous layer. To illustrate the dynamic expanding process, a diagram is shown below, to facilitate understanding.

As shown in the diagram above, there are 4 black nodes in the input layer, 5 black nodes in the hidden layer and 1 black node in the output layer. These black nodes along with all the black connections, stand for the NN that we construct, after seeing the first example. The blue node in the input layer represents the new feature we see in later examples. If we see a new feature, we create a new neuron and connect it to all neurons in the next layer. The red node in the output layer stands for the new label we see in examples later on. Similarly, if we see a new label, we create a new neuron and connect to all neurons in the previous layer. In both cases, we randomize the weights on all new connections and update the book keeping information stored in the internal `Neuroph`'s `MultiLayerPerceptron` class.

In the testing phrase, an interface `classify` function is provided from `Learner` class. It takes an integer

array of feature indices and a double array of feature values as arguments. We feed the features into the internal MLP instance to run through the NN and get the output from each output neuron. We map all outputs to find the index of the label category. The mapping from label to index is stored in `predictions` array instance in `Learner` class.

6.4 Evaluation

We evaluate the `MultiLayerPerceptron` learner in two aspects. First, we perform a simple overfitting test, to verify that the implementation is correct. Second, we evaluate it on Brown Corpus Data Set, in Appendix D.

6.4.1 Overfitting Test

We use Lenses Data Set (Refer Appendix C) [Lichman, 2013] from University of California, Irvine, as a simple data set to do overfitting test, which means we train `MultiLayerPerceptron` on the entire data set for a number of iterations and check if the classifier can evaluate the same data set all correctly.

The feature vector is a 9 bit one hot vector, thus the input layer has 9 neurons. Similarly, the output layer has 3 neurons. We construct the MLP with a single hidden layer, with 18 neurons.

After 100 iterations, MLP classifier classifies all examples with 100% accuracy.

6.4.2 Brown Corpus

We use Brown Corpus for context sensitive spelling correction (Refer Appendix D) [Golding and Roth, 1996], [Golding and Roth, 1999] as evaluation of `MultiLayerPerceptron` on sparse feature data sets.

We also compared `MultiLayerPerceptron` against `SparseAveragedPerceptron` and `SparseWinnow`, which are embedded in `SparseNetworkLearner` for multi-class classification.

The evaluation results, in terms of classification accuracy, are shown in Table 6.1. The parameters used for `SparseAveragedPerceptron` and `SparseWinnow` are listed in Table 6.2 and the parameters used for `MultiLayerPerceptron` are listed in Table 6.3. In particular, the learning rate for `MultiLayerPerceptron` in all cases is 0.2 and the training time and testing time listed in Table 6.3 are in seconds.

MLP vs SAP vs SW

As shown in Table 6.1, in most cases, `MultiLayerPerceptron` without hidden layers, performs slightly better. Notable examples are "among, between", "begin, being", "its, it's", "fewer, less", and "I, me".

No Hidden Layer vs Single Hidden Layer

In terms of performance, in the majority cases, `MultiLayerPerceptron` with one single layer is slightly better than `MultiLayerPerceptron` without any hidden layers. Notable examples are "amount, number", "cite,sight,site", and "its, it's". Most of the other cases, `MultiLayerPerceptron` with single hidden layer performs at least as good as `MultiLayerPerceptron` with no single hidden layers.

In terms of training time and testing time, as shown in Table 6.3, `MultiLayerPerceptron` with a single hidden layer takes much longer than its counterparts, `MultiLayerPerceptron` with no hidden layer. The magnitude of the number of connections is proportional to the number of nodes in the single hidden layer. We pick 100 and 80, mostly as the number of nodes in the hidden layer. This creates 100 times more connections in the larger NN. Thus, in turn, the training time and the testing time, grow up, proportionally.

Table 6.1: Brown Corpus Accuracy Table

Brown Corpus				
Confusion Set	Algorithms			
	MultiLayerPerceptron		SparseAveragedPerceptron	SparseWinnow
	No Hidden Layers	Single Hidden Layers		
accept, except	86	88	84	88
affect, effect	97.959	97.959	97.959	97.959
among, between	82.258	80.645	75.269	76.882
amount, number	81.301	84.553	75.61	74.797
begin, being	96.575	96.575	94.521	95.205
cite, sight, site	79.412	82.353	73.529	85.294
country, county	90.323	90.323	93.548	93.548
its, it's	96.721	97.268	94.809	93.716
lead, led	87.755	97.755	87.755	89.796
fewer, less	93.333	93.333	90.667	92
maybe, may be	95.833	94.792	93.75	94.792
I, me	98.857	98.857	97.959	97.714
passed, past	90.541	87.838	90.541	91.892
peace, piece	84	82	82	86
principal, principle	88.235	88.235	88.235	85.294
quiet, quite	95.455	93.939	92.424	93.939
raise, rise	84.615	84.615	82.051	84.615
than, then	96.304	96.541	96.693	96.304
their, there, they're	96.941	97.002	95.882	95.882
weather, wether	98.361	98.361	98.361	98.361
your, you're	97.326	97.861	93.048	95.513

Table 6.2: Brown Corpus Evaluation Parameters for SAP and SW

Brown Corpus							
Confusion Set	Algorithms						
	SparseAveragedPerceptron			SparseWinnow			
	LearningRate	Thickness	Iterations	LearningRate	Thickness	Beta	Iterations
accept, except	0.5	1.5	5	1.1	3	0.99009	5
affect, effect	0.1	2.5	5	2	1	0.5	5
among, between	0.1	1	5	2	4.5	0.99009	40
amount, number	0.5	1.5	30	1.005	1	0.99009	5
begin, being	0.1	1	5	1.1	1	0.99009	5
cite, sight, site	0.1	2	10	2	4.5	0.995	30
country, county	0.1	2	30	1.01	1	0.99009	5
its, it's	0.5	1.5	5	1.1	1.5	0.5	10
lead, led	0.1	1	10	1.1	2	0.99009	10
fewer, less	0.5	1	30	1.1	1	0.5	5
maybe, may be	0.1	1	5	2	3.5	0.909	10
I, me	0.1	1.5	5	2	4.5	0.5	10
passed, past	0.5	1.5	5	1.005	1	0.99009	5
peace, piece	0.5	1	30	1.01	3	0.5	5
principal, principle	0.005	1	20	1.01	1	0.909	20
quiet, quite	0.5	1.5	10	1.01	1	0.5	10
raise, rise	0.5	1	10	1.01	1	0.995	20
than, then	0.005	1	20	2	4	0.909	5
your, you're	0.5	1	5	1.005	1.5	0.909	10
weather, wether	0.5	2	5	1.005	3.5	0.995	10
your, you're	0.5	1.5	20	2	1	0.909	10

Table 6.3: Brown Corpus Evaluation Parameters for MLP

Brown Corpus							
Confusion Set	No Hidden Layer			Single Hidden Layer			
	Training Time	Testing Time	Iterations	Nodes	Training Time	Testing Time	Iterations
accept, except	2.119	0.012	10	80	49.914	0.070	100
affect, effect	1.980	0.006	100	80	66.170	0.086	100
among, between	10.095	0.021	100	100	1013.091	1.293	100
amount, number	5.006	0.019	100	100	352.638	0.448	100
begin, being	6.954	0.019	100	100	541.497	0.650	100
cite, sight, site	2.000	0.006	100	80	31.118	0.029	100
country, county	3.168	0.009	100	100	173.307	0.204	100
its, it's	27.013	0.055	100	100	2990.442	4.112	100
lead, led	2.209	0.008	100	80	61.065	0.083	100
fewer, less	4.838	0.008	100	100	307.670	0.318	100
maybe, may be	5.160	0.017	100	100	431.874	0.571	100
I, me	290.837	0.618	100	100	11737.291	13.793	100
passed, past	3.393	0.008	100	100	181.867	0.202	100
peace, piece	2.514	0.008	1000	80	82.042	0.086	100
principal, principle	1.919	0.005	1000	100	33.421	0.029	100
quiet, quite	2.798	0.009	100	100	149.441	0.192	100
raise, rise	1.738	0.009	100	100	26.519	0.050	100
than, then	55.658	0.120	100	80	5928.723	7.631	100
your, you're	212.378	0.349	100	100	9847.659	9.418	100
weather, wether	2.873	0.007	100	100	138.153	0.162	100
your, you're	9.514	0.002	100	100	1053.026	1.222	100

Chapter 7

Evaluation Across Algorithms

7.1 Algorithm Tests

We run multiple existing algorithms in LBJava on a simple, programatically generated, yet relatively large data set, from Appendix B, as algorithm tests.

The intent for these algorithm tests is to verify the correctness of the existing learning algorithms in LBJava. These algorithm tests serve as large unit tests and regression tests for learning algorithms. In the practice of software engineering, changes in the code base are constantly made. Regression tests and unit tests are practices to make sure that new changes made into the code, do not introduce bugs or issues. Although, there is no way to give a definite answer that there is no bugs in the system, but tests are critical measures to take, to prevent introducing or causing bugs to the greatest extent.

We have shown the class inheritance hierarchy diagram in Figure 7.1. *Learner* is the root of all learning algorithms, providing only abstract interface. *LinearThreshold* is also an abstract class, representing a large family of learning algorithms, including *Perceptron* and *Winnow*. All learning algorithms are embedded into *SparseNetworkLearner*, due to the SNoW architecture, as discussed in Section 3.3.

We have evaluated a list of learning algorithms, on Monotone Data Set, with parameters, $l = 10, m = 500, n = 1000, k = 50000$. The evaluation results and their associated parameters are shown in Table 7.1. Note that *PassiveAggressive* does not allow user configuration for its parameters. *LR* stands for learning rate and *Iter* is abbreviation for iteration. Moreover, *AdaGrad* and *SGD* come with two loss functions and each loss function is treated as a distinct learning algorithm, presented in the table.

From Table 7.1, we demonstrate that the existing learning algorithms behaves well, with almost 100% accuracy, on a clean and balanced simple, yet large data sets. These algorithms are hooked up onto *Semaphore*, a continuous integration build system, that would run these algorithms as tests, triggered by every single commit of changes.

Figure 7.1: LBJava Learning Algorithms Class Diagram

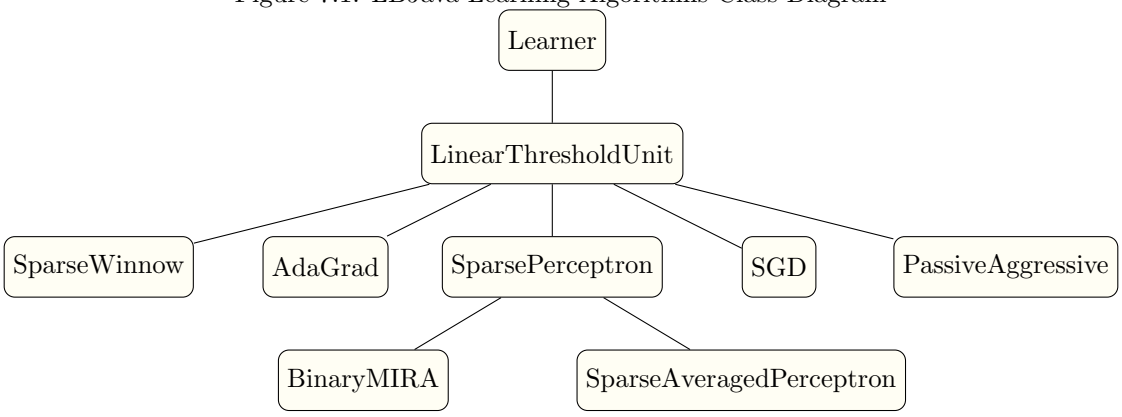


Table 7.1: Algorithm Tests on Monotone Data Set

Monotone Data Set		
	Accuracy	Parameters
SparseAveragedPerceptron	99.99	LR = 0.1; Iter = 10
SparseWinnow	99.44	LR = 1.01; Beta = 0.99; Iter = 10
SGDClassification Hinge	99.99	LR = 0.01; Iter = 10
SGDClassification LMS	99.25	LR = 0.001; Iter = 10
AdaGradClassification Hinge	99.72	LR =1; Iter = 10
AdaGradClassification LMS	100	LR = 1; Iter = 10
SparsePeceptron	99.22	LR = 0.1; Iter = 10
BinaryMIRA	100	LR = 1; Beta = 2; Iter = 10
PassiveAggressive	99.2	N/A; Iter = 10

Table 7.2: Evaluation of Learning Algorithms on Chunking

NLP Tasks Evaluation		
	Accuracy	Parameters
SparseAveragedPerceptron	97.292	LR = 0.1; TK = 2; Iteration = 50
SparseWinnow	96.277	LR = 1.01; TK =2; Beta = 0.909; Iteration = 50
BinaryMIRA	97.372	LR = 0.1; Beta = 1.0; Iteration = 50
PassiveAggressive	97.062	N/A; Iteration = 50
SparsePeceptron	97.216	LR = 0.1; TK = 2; Iteration = 40

7.2 Evaluation on NLP Task

We evaluate multiple algorithms in LBJava on, actual NLP tasks, particularly chunking. A chunker, [Punyakanok and Roth, 2001] also known as the shallow parser, partitions plain texts into sequences of semantically related words. The type for each partition is also assigned.

For example, sentence "Jack and Jill went up the hill to fetch a pail of water" is parsed as the following:

```
[NP Jack and Jill ] [VP went ] [ADVP up ] [NP the hill ]
[VP to fetch ] [NP a pail ] [PP of ] [NP water ] .
```

Chunking is simpler than full parsing, where a parse tree is generated indicating the nested structure, and it can serve as an aid for full parsers.

We present the evaluation results, using multiple learning algorithms, along with their associated parameters, in Table 7.2.

In Table 7.2, *LR* stands for learning rate, and *TK* is the abbreviation for thickness. Again, *PassiveAggressive* does not allow user configured parameters.

It is shown in Table 7.2 that the existing learning algorithms works fairly well on chunking, with roughly 50 iterations, given the training dataset of 211727 examples.

Chapter 8

Miscellany

8.1 Tutorials and Documentation

LBJava was first developed and implemented in [Rizzolo and Roth, 2007], [Rizzolo and Roth, 2010]. It is designed with the LBJ abstraction layer that takes away the low level interactions to develop ML systems, with simplicity. Yet, due to lack of a clear documentation and some internal bugs and issues, it is difficult to get started from user's perspective. From developer's perspective. it also proposes a steep learning curve on understanding the architecture, pipeline, debugging, and further development, since the code base is relatively large, about 500 Java class files and more than 50,000 lines of code.

To facilitate new users to get started on using LBJava faster and smoothly, such as running a demo example, setting up LBJava in a new project, a tutorial for new users is added to the code repository.

On the other hand, all the existing tutorials and examples are for classification. Regression is a fairly new domain in LBJava. A detailed tutorial with guidelines and a demo example is added to the set of examples.

Moreover, we recently realized to avoid using learning algorithms for binary classification directly. All learning algorithm should be used embedded inside `SparseNetworkLearner`, as discussed in Section 3.3. Documentation is updated on this point, in the code repository.

In additional, documentation on operators [] and %, as discussed in Section 3.2 and their usage scenarios and effects is also added.

Furthermore, due to the rather complicated class hierarchies and relationships, a class diagram, with inheritance relationship is added, to facilitate understanding.

Lastly, documentation on parameter tuning, as shown in Section 3.2.3, is also added.

8.2 Compilation Script

The compiling script for all LBJava examples, has issues when compiling or generating code for all examples, after the first time. The script falls into a dead state, not executing forward. To clean every LBJava generated

file, before re-running the script is not efficient. This issue is resolved, by adding a flag `-x` to the compiling arguments.

The original compilation script only generates code for all examples. We add an argument from command line, taking a string, such as "all", "regression", "spam" or "newsgroup", to compile each example individually.

From root of the project repository, run the following commands:

```
cd lbjava-examples
sh compileLBJ.sh <argument>
```

where

```
<argument> = all, badges, entity, newsgroup, sentiment, setcover, regression
```

Chapter 9

Conclusion and Future Work

We have presented in this thesis on the work contributed to LBJava.

Firstly, regression evaluation metrics, with system and statistical information is added. Second, we introduce AdGrad, for both classification and regression, to LBJava and extend SGD for classification. We compare SGD and AdaGrad, in classification and regression scenarios. Third, we evaluate across multiple learning algorithms in LBJava, on the same programmatically generated datasets. Fourth, we introduce NN, in particular MLP to LBJava and compare with other learning algorithm. Fifth, we evaluate and compare across multiple learning algorithms on NLP tasks, i.e. POS Tagging and Chunking. Lastly, we fix some issues and add multiple documentation and tutorials on several domains, to facilitate understanding.

For future work, we propose to introduce parallelism into LBJava, particularly for `SparseNetworkLearner`. It would speed up the training process, especially, if the number of labels is large. The other direction is to add more NN learning algorithms, besides the basic MLP.

References

[Blum, 1990] Blum, A. (1990). Learning boolean functions in an infinite attribute space. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 64–72. ACM.

[Carlson et al., 1999] Carlson, A., Cumby, C., Rosen, J., and Roth, D. (1999). The snow learning architecture. Technical report, Technical report UIUCDCS.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159.

[Fanaee-T and Gama, 2014] Fanaee-T, H. and Gama, J. (2014). Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, 2(2-3):113–127.

[Foundation, 2004] Foundation, T. A. S. (2004). Apache license, version 2.0.

[Golding and Roth, 1996] Golding, A. R. and Roth, D. (1996). Applying winnow to context-sensitive spelling correction. In *ICML*, pages 182–190.

[Golding and Roth, 1999] Golding, A. R. and Roth, D. (1999). A winnow based approach to context-sensitive spelling correction. *Machine Learning*, 34(1-3):107–130.

[Kordjamshidi et al., 2015] Kordjamshidi, P., Roth, D., and Wu, H. (2015). Saul: Towards declarative learning based programming. In *IJCAI*.

[Lichman, 2013] Lichman, M. (2013). UCI machine learning repository.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

[Punyakanok and Roth, 2001] Punyakanok, V. and Roth, D. (2001). The use of classifiers in sequential inference. In *NIPS*, pages 995–1001. MIT Press.

[Rizzolo, 2011] Rizzolo, N. (2011). Learning based programming. Technical report.

[Rizzolo and Roth, 2007] Rizzolo, N. and Roth, D. (2007). Modeling discriminative global inference. In *ICSC*, pages 597–604, Irvine, California. IEEE.

[Rizzolo and Roth, 2010] Rizzolo, N. and Roth, D. (2010). Learning based java for rapid development of nlp systems. In *LREC*, Valletta, Malta.

[Roth, 1999] Roth, D. (1999). Learning based programming.

[Roth, 2005] Roth, D. (2005). Learning based programming. *Innovations in Machine Learning: Theory and Applications*.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA.

Appendix A

Bike Sharing Data Set

Bike Sharing Data Set [Lichman, 2013] [Fanaee-T and Gama, 2014] is maintained by University of California in Irvine.

There are two sets of data sets in this Bike Sharing Data Set, namely Day and Hour.

This is a regression data set containing 17379 instances with 14 features in Hour data set and 731 instances with 13 features.

Attributes information is listed below. Day data set does not have the hour attribute.

1. Frequency, in Hertzs.
2. instant: record index
3. dteday : date
4. season : season (1:springer, 2:summer, 3:fall, 4:winter)
5. yr : year (0: 2011, 1:2012)
6. mnth : month (1 to 12)
7. hr : hour (0 to 23)
8. holiday : weather day is holiday or not (extracted from [Web Link])
9. weekday : day of the week
10. workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
11. weathersit
    (a) Clear, Few clouds, Partly cloudy, Partly cloudy
    (b) Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
    (c) Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds

(d) Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog

12. temp : Normalized temperature in Celsius.

13. atemp: Normalized feeling temperature in Celsius.

14. hum: Normalized humidity. The values are divided to 100 (max)

15. windspeed: Normalized wind speed. The values are divided to 67 (max)

16. casual: count of casual users

17. registered: count of registered users

18. cnt: count of total rental bikes including both casual and registered

Appendix B

Monotone Data Set

Monotone Data Set is a programmatically generated, artificial data sets and as the name implies, consists of only 0s and 1s. The motivation to develop such data set is that, we would like to verify the correctness of all existing learning algorithms in LBJava, using a simple, yet relatively large data set. This also serves as a purpose of regression testing, that every change made in the code of all existing learning algorithms, has to pass the correctness verification step, before merging. The reason to generate such data set programmatically is that we do not want to store large data sets in the code repository as it intrinsically increases the size of the code base.

The interface for constructor of this data set class is shown in the following:

```
public AlgoDataSet(int l, int m, int n, int k)
```

where l is the number of active attributes, m is the dimensionality of set of attributes where actives attributes are randomly chosen, n is the total number of attributes, k is the total number of examples. We follow the conventional rule to split such data set into 80% for training and 20% for testing.

For this data set, half of the examples are positive example and the other half are negative examples.

For positive examples, we pick randomly and uniformly of l attributes from $x_1, \ldots, x_m$ and set them to 1. We then set the other $m - l$ attributes to 0 and the rest of the $n - M$ attributes to 1 uniformly with probability of 0.5.

For negative examples, we pick randomly and uniformly of $l - 2$ attributes from $x_1, \ldots, x_m$ and set them to 1. We then set the other $m - l + 2$ attributes to 0 and the rest of the $n - m$ attributes to 1 uniformly with probability of 0.5.

Lastly, upon the generation of all example, we shuffle all examples so that they appear randomly sequentially.

Appendix C

UCI Lenses Data Set

Lenses Data Set [Lichman, 2013] is maintained by University of California in Irvine.

It is a simple data set containing only 4 attributes, with categorical values, and 3 classes. There are only 24 instances.

Attributes information is listed below.

1. animal name: string
2. age of the patient: (1) young, (2) pre-presbyopic, (3) presbyopic
3. spectacle prescription: (1) myope, (2) hypermetrope
4. astigmatic: (1) no, (2) yes
5. tear production rate: (1) reduced, (2) normal

Classes information is listed below.

- 1 : the patient should be fitted with hard contact lenses
- 2 : the patient should be fitted with soft contact lenses
- 3 : the patient should not be fitted with contact lenses

Appendix D

Brown Corpus Data Set

Brown corpus for context sensitive spelling correction [Golding and Roth, 1996], [Golding and Roth, 1999] is a data set containing words which are context sensitive. For example, "accept" vs "except"; "affect" vs "effect"; "raise" vs "rise" and "their" vs "there" vs "they're".

`.feat` files are in the SNoW format [Carlson et al., 1999]. Each line corresponds to an example. The class label is the first value, taking values from 0 to $k-1$, where k is the number of possible labels. Feature values are all greater than k and they are indices of words, in ascending order.

All data sets are already divided into 80% for training and 20% for testing.

There are in total 21 different sets of context sensitive spelling words.

1. accept, except
2. affect, effect
3. among, between
4. amount, number
5. begin, being
6. cite, sight, site
7. country, county
8. its, it's
9. lead, led
10. fewer, less
11. maybe, may be
12. I, me

13. passed, past

14. peace, piece

15. principal, principle

16. quiet, quite

17. raise, rise

18. than, then

19. their, there, they're

20. weather, whether

21. your, you're

Appendix E

UCI Zoo Data Set

Zoo Data Set [Lichman, 2013] is maintained by University of California in Irvine.

It is a simple data set containing 15 attributes, with boolean and numerical values, and 7 classes. There are 101 instances.

Attributes information is listed below.

1. animal name: string
2. hair: boolean
3. feathers: boolean
4. milk: boolean
5. airborne: boolean
6. aquatic: boolean
7. predator: boolean
8. toothed: boolean
9. backbone: boolean
10. breathes: boolean
11. venomous: boolean
12. fins: boolean
13. legs: numerical (set of values: 0,2,4,5,6,8)
14. tail: boolean
15. domestic: boolean

16. catsize: boolean

17. type: numerical (integer values in range [1,7])